Design and Implementation of a high performance IPC using Socket API

Bachelor Thesis

by

Daniel Aeneas von Rauchhaupt

University of Potsdam
Institute for Computer Science
Operating Systems and Distributed Systems

Supervisors:
Prof. Dr. Bettina Schnor
Max Schrötter

Potsdam, July 22, 2024

von Rauchhaupt, Daniel Aeneas
rauchhaupt@uni-potsdam.de
Design and Implementation of a high performance IPC using Socket API
Bachelor Thesis, Institute for Computer Science
University of Potsdam, July 2024

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, den 22. Juli 2024

Daniel Aeneas von Rauchhaupt

Abstract

This is an abstract which briefly summarizes the key points of the bachelor thesis.

Deutsche Zusammenfassung

Dies ist eine Zusammenfassung welche die Schlüsselpunkte der Bachelorarbeit kurz beschreibt.

Contents

1 Introduction

Development in internet infrastructure has led to higher interconnectedness over the course of the last century. Interacting with companies and governmental facilities is primarily done online. Deploying servers, providing specific services to the public, has become common place in daily life. With a continually growing importance of these servers, exploiting them becomes attractive for malicious actors. Identifying threats and security breaches is required to provide consistent uptime of servers. While traditional firewalls provide a starting point in defense against exploitation, they are not impenetrable. Common attacks, such as a Denial-of-service (DoS) attack, can circumvent these measures by imitating genuine clients. Draining system resources by sending illegitimate communication requests is simultaneously easy to do, and hard to protect against.

Additional monitoring of incoming traffic is done by Intrusion Detection Systems (IDSs). Interpreting a clients intent is done by analyzing by system logs, as well as system and network resources. To actually combat incoming attacks, a Intrusion Prevention System (IPS) is required which actively defends system resources. A common industry standard for such an IPS is Fail2ban[1]. It scans a variety of information available on the host system, predominantly using log files as its source, technically making it a Host-based Intrusion Prevention System (HIPS). Clients deemed a threat to the host system are prohibited to interact with the host. Fail2ban achieves this by modifying the systems firewall.

However, previous work has shown that Fail2ban does not scale well when having to process a large number of log files instantaneously[2]. For networks with high bandwidth, a sudden influx in log messages can indicate an ongoing attack on the system. Fail2ban being unable to efficiently perform its duties in these heavy-load scenarios has serious downfalls. This inconsistent performance makes the system vulnerable against DoS attacks. It was determined that the intrinsic file-based logging approach of Fail2ban does not provide the necessary bandwidth or low latency required to repel DoS attacks successfully.

To remedy this issue a light-weight alternative to Fail2ban was developed: Simplefail2ban[3]. While inheriting the basic functionality of Fail2ban, this application provides to option to replace slow file-based logging with alternative Inter-process communication (IPC). During development, a shared memory IPC type was implemented. This allowed Simplefail2ban to outperform Fail2ban effortlessly[3], but if better alternatives exists remains unclear.

The main goal of this thesis it to design and implement an IPC mode based on unix domain sockets into Simplefail2ban in order to protect against DoS attacks. This includes an easily usable Application Programming Interface (API) and the option to attach multiple reader processes to the IPC architecture. In order to evaluate the performance of this socket IPC, a comparison with the already implemented shared memory and file IPC types is conducted.

Firstly, this thesis introduces background information regarding both Fail2ban and Simplefail2ban. An explanation of the basic concepts used for the IPC is also included. Following that, a chapter is dedicated to introducing the design of the unix socket IPC architecture. A separate chapter will explain the intricacies of the implementation of said IPC type. In addition, the design of all experiments will be explained. To determine the performance of the socket IPC, an extensive evaluation of the conducted measurements is included in this thesis. Finally, a summary of the findings of this thesis and a verdict on performance of both the shared memory and socket IPC type for Simplefail2ban is included.

2 Background & Motivation

This section establishes a definition for Host-based intrusion detection/prevention systems and introduces the example Fail2ban. An introduction to an alternative solution, Simple-fail2ban, and its necessity will also be discussed. Lastly, any external tools used in this thesis will also be discussed.

2.1 Host-based intrusion detection and prevention

Intrusion detection and prevention systems are tasked with monitoring the system and ensuring that no threat is present. While the former is only tasked with detecting on-going attacks, the latter actively defends system resources from exploits. The restriction to only utilize data available on the host system, differentiates a Host-based Intrusion Detection System (HIDS) from other forms of Intrusion Detection System (IDS). In general, collecting and analyzing data, identifying outliers, evaluating the risk these outliers pose, and responding to any potential threats or unusual behavior to minimize potential harm to the system, is the main task of an Intrusion Prevention System (IPS). According to James P. Andersons study "Computer security threat monitoring and surveillance"[4] a threat is any deliberate attempt to either

- access data,
- manipulate data, or
- render a system unreliable or unusable.

With the ever-present risk of a system having a previously unknown vulnerability, proactive measures must be taken to prevent malicious actors' exploits. Real-time intrusion detection systems are required to achieve this goal. The motivation for such a system is outlined by Dorothy E. Denning[5]:

- The majority of systems have vulnerabilities, rendering them susceptible.
- Replacing systems with known vulnerabilities is difficult. Specific features may only be present in the less-secure system.
- Developing absolutely secure systems is difficult, since the explicit absence of vulnerabilities can rarely be proven.
- Secure systems remain vulnerable to insiders misusing their privileges.

For the purposes of this paper, defending against a Denial-of-service (DoS) attack, the basic assumption that any system is exploitable will suffice.

A HIDS generally collects data from multiple sources, freely provided by the host. Such auditing of data needs to be tamper-proof and non-bypassable. Low-level system calls, often containing such data, are preferable to achieve this goal. The anomaly based approach allows an intrusion detection system to create profiles representing legitimate behavior of clients, users and applications. Using statistical tests on normal behavior of clients, any deviations are detected and interpreted as an attack on the system. This retains the advantage of not explicitly defining attack patterns, creating a more robust system which can identify new threats on its own.[6]

2.1.1 Fail2ban

Fail2ban is an open-source intrusion prevention system, developed in Python, running in the user space level. In contrast to an intrusion detection system, an IPS, such as Fail2ban, immediately takes deliberate measures once a threat has been identified to stop attacks on a system. By default, Fail2ban scans a variety of commonly used log files using Regular Expression (Regex), also called filters, to identify threats. It is therefore able to parse and monitor log data of a variety of different applications. A client will be identified as a threat if it repeatedly fails a certain task, for example establishing a Transmission Control Protocol (TCP) connection. Such a client is then banned by modifying the system firewall, addding its Internet Protocol (IP) address, to deny any further incoming traffic.[1]

In detail, Fail2ban creates so called jails. These jails are saved on persistent storage. Therefore, restarting Fail2ban or the machine running it will not result in a loss of current jail entries. A jail consists of a log path, a certain filter, an action and a variety of customizable parameters. The filter requires at least one Regex pattern. These patterns define what behavior Fail2ban should tolerate or not. An action, commonly a command or program, is to be executed once a client has been deemed a threat. Further parameters define the time the action will be active (ban time) and how often bad behavior of a client must be identified (ban limit) in log files to issue a ban. In practice, if a client fails to adhere to what the filter of a jail defines as proper behavior, vital information of that client is deduced by the analyzed log messages. This includes the IP address of the client. A ban will then be issued and a certain action, for example dropping all traffic with the source IP of the banned client, would be performed. To issue such a ban, temporary changes to the system firewall, using iptables, are performed. iptables allows user space programs, such as Fail2ban, to modify, add and remove rules for packet filtering. An incoming package has to pass each set of rules before reaching the destined application. Fail2ban creates a separate rule for each banned client via iptables. New incoming packets are checked against all rules defined by iptables, or until they infringe upon at least one rule. Especially when many clients need to be banned this exerts an ever increasing load on the processing capabilities of the firewall, as each banned client corresponds to one additional rule future traffic has to be compared to.[2]

2.1.2 Extended Berkeley Packet Filter

The extended Berkeley Packet Filter (eBPF) provides the opportunity to run user-generated code in a privileged setting, such as the kernel. Such eBPF programs are written in high-level programming languages, for example C. Compilers convert these programs to eBPF bytecode in user space. Successfully deploying the code requires an eBPF verifier to accept the program. This is done exclusively in kernel space as to limit risk to the security of the operating system. If the eBPF program is accepted, the program will be converted to eBPF native machine code. There are several hooks to which an eBPF program can be attached to. Depending on the chosen hook, the eBPF program is deployed in or even before the network stack. Meaning, the eBPF program receives incoming traffic while the operating system is still processing it in kernel space.[2]

In this thesis, the eXpress Data Path (XDP) Driver hook is used for all eBPF programs. Simply put, the eBPF program and its user-generated code is run before the kernel has performed its usual processing steps for incoming traffic. This way, the program will receive each incoming packet and can decide to let it pass to the kernel unhindered, or drop it at an early processing stage.

Since eBPF programs are event-driven, they only handle one packet at a time. In order to communicate with other programs or even store information, eBPF Maps are used. These maps are a key-value store and provide persistent storage. However, the size of eBPF maps needs to be defined before runtime, as it cannot be altered at a later stage.[2]

Using eBPF programs provides a significant advantage over the iptables approach of filtering packets. It is possible to drop unwanted packets before they reach the computation heavy kernel network stack, potentially saving resources on packets which ultimately would have been discarded anyway. And while eBPF programs have a variety of useful other applications, for purposes of this thesis, they are only used to either accept packets and pass them to the kernel or drop them to lighten workload.

2.1.3 Simplefail2ban

Florian Mikolajczak has shown[2] that Fail2ban performs poorly when dealing with large amounts of incoming unwanted traffic. This issue remains even after an alternative, and competitive, method of filtering incoming traffic using eBPF programs was implemented. To remedy this shortcoming, Simplefail2ban was developed[3]. It was suspected that Fail2ban is losing performance by exclusively utilizing traditional file-based logging. The goal was to implement an IPS that can prohibit malicious actors from sending traffic to the host system, similarly to Fail2ban, without having to rely on file-based logging.

Simplefail2ban provides the option to use a shared memory section to receive log messages. This significant change proved to be a faster method to transmit log messages from an application directly to Simplefail2ban. However, the general requirements for banning a client are unchanged. The IPS still monitors incoming log messages for disallowed behavior.[1] Each violation of the rules imposed by Simplefail2ban results in the clients IP being logged in a hashtable. If the number of entries for one IP address surpasses the defined ban limit, that client is banned via one of the banning threads of Simplefail2ban. This ban is facilitated by adding the IP address to a list of banned clients with the current timestamp, and an eBPF map. An eBPF program developed by Florian Mikolajczak will check if incoming traffic should either be dropped or passed along to the kernel, depending on the eBPF map entries[2]. The list of banned clients is routinely checked by the unbanning

[1] Since Simplefail2ban is just a prototype, the distinction between allowed and disallowed behavior is based upon the payload of incoming traffic.

thread, removing clients whose ban time has elapsed from the hashtable, ban listn and eBPF map, effectively re-allowing client interaction.[3].

2.2 Inter-process communication

While a variety of methods for inter-process communication exist, the nature of this thesis only necessitates the detailed comparison between the shared memory and socket approach. Therefore, understanding technical details of both Inter-process communication (IPC) types is vital to reach a conclusive verdict. Development was conducted on a Linux based system which will be reflected when discussing technical details.

2.2.1 Shared memory approach by Paul Raatschen

While Paul Raatschen initially considered multiple IPC types, such as shared memory, named pipes, sockets and message queues, only the shared memory approach was implemented as the most viable option. This was because it did not require any involvement of the kernel during write or read operations, and thus no context-switches between kernel- and user-space. Hence, if the synchronization overhead for the communication processes could be kept to a minimum, the IPC could operate almost at the speed of normal memory access. With no precursor on how to implement IPC based on shared memory, Paul Raatschen settled for an accumulation of independent segments. Each segment consists of a single ring buffer.[3]

Ring buffers are common array-like data structures. When saving data in a ring buffer, data is written in order into the buffer. For each data entry, the writer index position is incremented by one. Once the buffer is filled, the writer index loops back to the beginning of the array. Receiving data from a ring buffer works in a similar fashion. Once the end of the array is reached, the reader index is again set to the beginning of the ring buffer. Therefore, one can imagine the end of a ring buffer being connected with its first array element, resulting in a circular array. Overall, this results in data being read in a first-in first-out manner, with the index of the writing process preceding the index of the reading process. However, due to a multitude of reasons, the writer process might catch up to the index of the reader process. If this happens, there are two possible courses of action: Either wait for the reader index to move, and then write new data into the ring buffer; or overwrite the entry not yet read by the reader process. While overwriting the entry in the ring buffer leads to loss of data, the writer process is not slowed down by the reader process. Using shared memory, the desired approach can be defined by setting the option "overwrite" to accept data losses[3].

Segments are defined via a global header, dictating certain shared variables. This includes the number of ring buffers, the number of entries each ring buffer has and the size of each array element in byte. While other components exist in the global header, they all serve to synchronize writers and readers in one way or another and are not vital in understanding the general design of the shared memory IPC type; for more details, refer to [3].

Once the shared memory section has been established, multiple reader processes can attach one reading thread to each segment. Yet, per design, only one writing thread attaches to each segment. This one-to-one mapping ensures no further synchronization between multiple writer threads is required. Sending and receiving data can now be performed by each thread individually according to the base principles of ring buffers outlined above.

2.2.2 Unix Domain Sockets

In order to explain what a unix domain socket is, one must understand regular internet sockets. On a Linux system, a socket is a file descriptor referring to an endpoint for communication[7]. While a variety of socket types exist, the actual socket (or file descriptor representing a socket) does not change. Instead, the way data is transmitted via a particular socket defines the socket type. The most common types of sockets are stream and datagram sockets.

Stream sockets provide a reliable two-way connection between communication partners. Not only do they guarantee that any data sent is transmitted without errors, but they also do preserve the order in which the data was sent. This behavior is achieved by utilizing the TCP.[8]

The foundation of TCP is a three-way handshake in which participants negotiate the parameters required for the data exchange. Error checking is performed on all messages. If data is corrupted, the recipient can and will request retransmission of the same data. A number of additional factors contribute to the complexity of TCP. However, for this thesis, the knowledge that TCP's reliability is achieved via the cooperation of all participating partners will suffice.

In contrast, datagram sockets, also called connectionless sockets, are considered unreliable, because it is based on the User Datagram Protocol (UDP), not TCP. UDP does not guarantee that data will arrive at its destination. Consequently, the reception of data in correct sequence cannot be guaranteed either. The lack of a reliable connection between communication partners, instead using a best-effort service, results in lower latency during data exchange.[8]

When a socket is only represented via a path name on a local system, it is called a unix domain socket (also known as AF_UNIX). Unlike stream or datagram sockets, unix domain sockets are used for local only inter-process communication. Therefore, while they do inherit similar functionality as the internet sockets, they can shed slow communication protocols and provide faster communication. Data is never sent beyond system boundaries and only handled by the kernel. TODO: unix -> UNIX with link to acro There are three socket types in the UNIX domain[9]:

- SOCK_STREAM: Stream-oriented socket (comparable to stream sockets), establishing connections and keeping them open until explicitly closed by one communication partner.
- SOCK_DGRAM: Datagram-oriented socket (comparable to datagram sockets), preserving message boundaries. In contrast to datagram sockets, SOCK_DGRAM is reliable and does not reorder sent data in most UNIX implementations.
- SOCK_SEQPACKET: Sequence-packet socket, is connection-oriented, preserves message boundaries, and retains the order in which data was sent.

In conclusion, unix domain sockets retain the flexibility provided by traditional internet sockets allowing for decreased latency, but at the cost of being bound to the local system.

2.3 Packet generator: TRex

TRex is an open source traffic generator developed by Cisco Systems, capable of generating both stateless and stateful traffic[10].

TRex is based on the Data Plane Development Kit (DPDK), which is a framework designed to increase packet processing speeds for a limited number of Central Processing Unit (CPU) architecture. The increase in performance is mainly attributed to the Poll Mode Drivers (PMDs), which bypass the kernel's network stack.[11]

Providing the ability to use multiple cores to generate traffic, TRex can send up to 200Gb/sec with hardware supported by the DPDK framework. Utilizing Scapy, a packet manipulation library written in Python[12], TRex is able to generate a customizable stream of traffic, allowing the user to modify any packet field.[10]

This feature will be used to modify the source IP of all generated packets, to simulate attacks involving a large number of clients.

The failure to achieve advertised traffic rates when using stateful traffic in certain scenarios was already observed by Paul Raatschen. When deploying Simplefail2ban, incoming traffic of banned clients is dropped by the IPS before reaching the network stack of the kernel. Therefore, no application receives any packets, and consequently, no reply is sent. This results in a loss of performance for TRex, as it expects an Acknowledgment (ACK) packet when sending a Synchronize/Acknowledge (TCP-SYN) packet.[3]

Therefore, in the scope of this thesis, TRex is used to generate UDP traffic only.

3 Design

The following chapter discusses the design of the socket Inter-process communication (IPC) for Simplefail2ban. While reasoning the choice of unix domain sockets, advantages and disadvantages are presented.

3.1 Reasoning for Unix Domain Sockets

In order to make an informed decision on which IPC type is suited best, requirements need to be specified. Since the task at hand is to defend against Denial-of-service (DoS) attacks on the host system, the following aspects are considered[3]:

- **Low latency**
  Responding quickly to incoming threats is key to successfully block incoming attacks. A quick transfer of data to the Intrusion Prevention System (IPS) facilitates faster banning of malicious attackers, before they can overwhelm the system. In general, low overhead is required to achieve these goals.

- **High bandwidth**
  Considering that the host system is bombarded with millions of packets each second during an ongoing DoS attack, high bandwidth between processes using the IPC is crucial. To avoid bottlenecks between the writer and the IPS, large amounts of data need to be transmissible at once instead of requiring separate transmissions.

- **Reliability**
  Ensuring that no crucial log messages are lost due to unreliability is desirable. Repeatedly missing information about malicious clients delays the response time of the IPS, risking uptime of the system and its services.

- **Scalability**
  Log messages can come from a multitude of sources and contain a variety of information. Multiple applications should be able to submit log messages to the IPS at once, and retain the possibility of providing it to other applications. Therefore, the option to have both multiple readers and writers should be present. While not necessary for the development of a host-based IPS, the option to scale beyond the local filesystem is interesting.

- **Portability**
  Developing an IPS requires it to actually be usable with already existing applications. Whereas this thesis aims at presenting a fully functioning proof of concept, potential future development may still require some flexibility. A well defined Application

> Programming Interface (API) that can realistically be integrated into any application, without the need for specific hard- and software, greatly facilitates portability.

Initially, the decision by Paul Raatschen to utilize shared memory as the IPC[3]. No other IPC method was implemented. To ensure that the shared memory approach is the most viable, an alternative needed to be chosen to be measured against.

The choice fell on unix domain sockets, due to the already existing write and read API and its support in the Posix standard on all UNIX systems since at least 2007[13]. Additionally, the utilization of sockets provides a great deal of flexibility during usage of the IPC. Since unix domain sockets are already an established mode of IPC, it is suspected that they also provide the desired low latency and high bandwidth. Furthermore, they are reliable[9] and theoretically provide the opportunity to replace them with traditional internet sockets to scale beyond the local system.

3.2 Design and abstractions

During the design process of the socket architecture it was decided that supporting attachment of multiple readers was essential, as already discussed in section 3.1. Each reader should receive all log messages sent by any writers. These restrictions led to the design illustrated in figure 3.1.

This figure displays the general data flow using the socket IPC type. Each reader application creates its own unix domain socket. Sockets are bound to a filesystem pathname. Readers can receive data from their own socket without having to compete or synchronize with other readers for data thanks to the one-to-one mapping between socket and reader. Meanwhile, writers can also independently write data into sockets without having to communicate with other writer processes. In order to guarantee that readers receive all data being sent via the socket IPC architecture, writers need to periodically recheck for newly opened sockets and always send their data to all available sockets. This results in the writers having a significant portion of the overhead, needing to resend identical data multiple times. Minimizing overhead on the reader side is important to maximize the limited computational time that crucial services, such as the IPS Simplefail2ban, have available to process incoming log messages.

To preserve the integrity of log messages, the unix domain socket needs to retain message boundaries, ruling out the SOCK_STREAM unix domain socket presented in section 2.2.2. Without the absolute guarantee of reliable behavior in regard to reordering of messages - only present in most UNIX implementations, but not all[9] - SOCK_DGRAM is unappealing as well. Therefore, the socket type choice falls on SOCK_SEQPACKET, a connection-oriented option that retains message-boundaries and sequence.

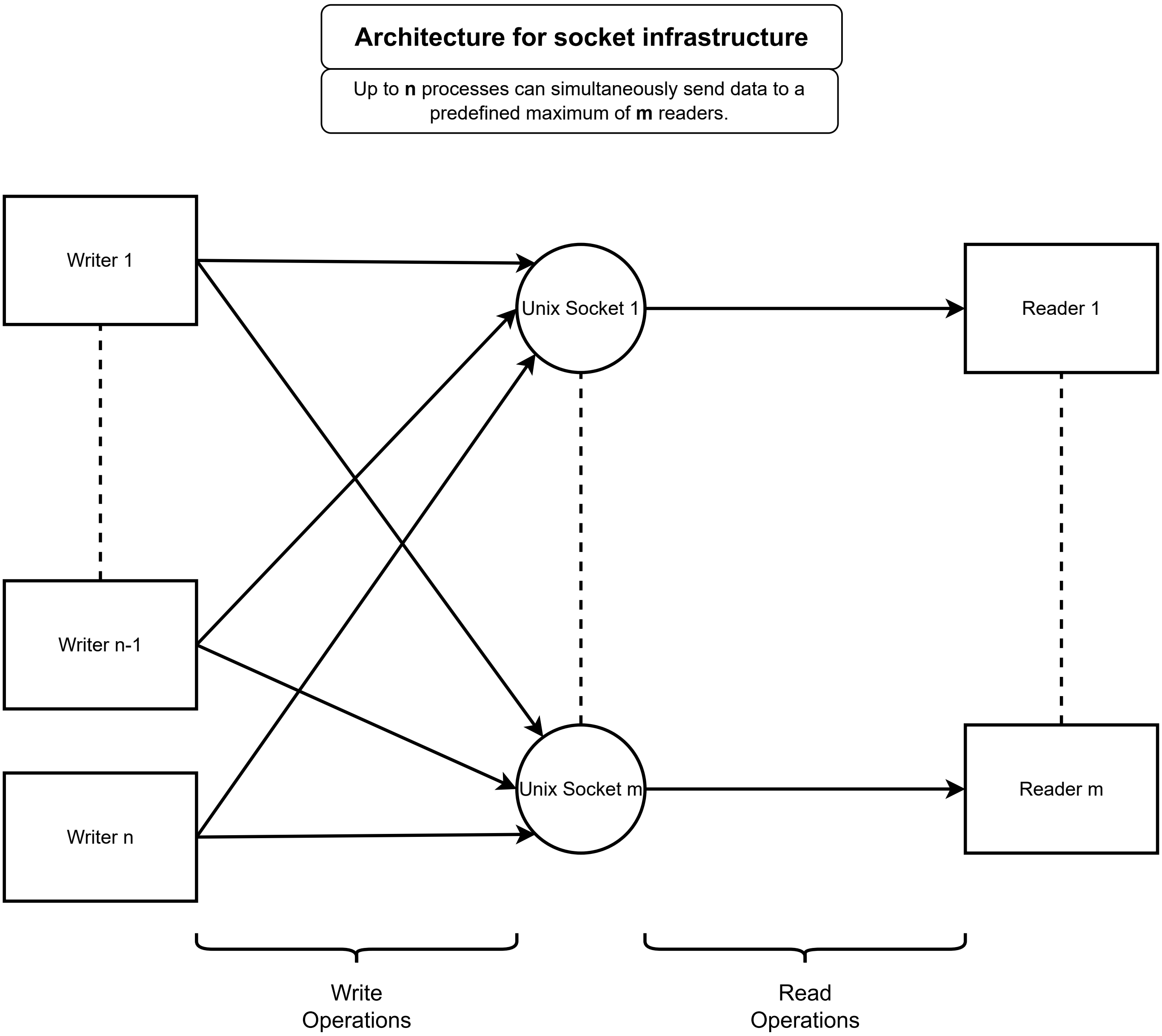

Architecture for socket infrastructure
Up to **n** processes can simultaneously send data to a predefined maximum of **m** readers.
Writer 1
Writer n-1
Writer n
Unix Socket 1
Unix Socket m
Reader 1
Reader m
Write Operations
Read Operations

Figure 3.1: Architecture for a n-reader and m-writers scenario using unix domain sockets.

4 Implementation

In this chapter, the implementation of the previously discussed design (Reference: 3.2) for the unix domain socket architecture in the programming language C is presented. This includes an explanation of the write and read Application Programming Interface (API). All error codes are numeric and defined in the file `io_ipc.h`.

4.1 Auxiliary functions and structures

When utilizing the socket Inter-process communication (IPC) type, some shared resources seen in 4.1 need to be set up. None of these are modifiable during runtime.

```c
#define MAX_AMOUNT_OF_SOCKETS 32

// This has to be long enough to fit the number or the socket and a terminating null byte
#define SOCKET_TEMPLATE_LENGTH 128
#define SOCKET_NAME_TEMPLATE "/tmp/unixDomainSock4SF2B_"
```

Algorithm 4.1: Parameters shared between readers and writers.

To ensure that no writer gets stuck continually checking for an infinite amount of new sockets, the global variable `MAX_AMOUNT_OF_SOCKETS` is defined as an upper limit. A special feature of the socket IPC type is the possibility of attaching a variable number of reader and writer processes, even during runtime. In fact, there is no actual limit for attaching new writer processes. Meanwhile, only up to `MAX_AMOUNT_OF_SOCKETS` reader processes can exists because of the strict one-to-one mapping between readers and sockets.

All unix domain sockets were bound to the filesystem, resulting in a common path to the location of all sockets needing to be supplied to both readers and writers. However, this `SOCKET_NAME_TEMPLATE` is not the full path to each socket. During runtime, each reader process trying to attach will append this name template with their own reader Identification (ID). The reader ID is determined by claiming the first ID not already in use. Since the length of the reader ID being appended to the `SOCKET_NAME_TEMPLATE` can vary, a length for this template is defined in `SOCKET_TEMPLATE_LENGTH`. It should be defined in such a manner, that both the appended reader ID and a terminating null byte can be appended to `SOCKET_NAME_TEMPLATE`.

Separating functions utilized by readers and writers results in an unwieldy API. Shared usage of functions by both sides is achieved by supplying function calls with the role of the calling process, either `SOCK_WRITER` or `SOCK_READER`.

```
int sock_init(union sock_arg_t *sock_args,
                  int role);
```

Algorithm 4.2: Initialization function for both reader and writer processes.

Therefore the function initializing communication between processes, `sock_init` as per 4.2 only requires a structure of parameters and the role of the calling process. Defining a union containing both writer and reader structures, as seen in 4.3, allows the user of the API to provide either one as a parameter for the same function. The actual purpose of `sock_init` is to enable connection between writer and readers by initiating the associated structure passed in the parameter `sock_args`. An explanation of both the `sock_writer_arg_t` and `sock_reader_arg_t` will follow in the next sections 4.2 and 4.3. Writers are provided with a list of possible locations of unix domain sockets belonging to reader processes. Meanwhile, readers are assigned a path, in which they create a unix domain socket. This path has to conform with `SOCKET_NAME_TEMPLATE` as outlined above. All sockets are set to be of the type SOCK_SEQPACKET.

```
union sock_arg_t{
     struct sock_writer_arg_t wargs;
     struct sock_reader_arg_t rargs;
};
```

Algorithm 4.3: Union containing either the parameters of a writer or reader process.

While other IPC types such as shared memory required an orderly detachment of writers and readers, this is not necessary for the socket approach. Instead, when terminating a reader process, only closure of the corresponding unix domain socket is necessary. Currently, stopping a writer process results in deconstructing the entire unix domain socket architecture. This results in the functions `socket_finalize` and `socket_cleanup`, as shown in 4.4 and 4.5 respectively, being identical in behavior. In fact, `socket_finalize` simply calls `socket_cleanup` and was only provided in the socket API to make a seamless replacement of other finalize-style functions when switching IPC types possible.

```
int sock_finalize(union sock_arg_t *sock_args, int role);
```

Algorithm 4.4: Cleanup initialization function of socket IPC.

```
int sock_cleanup(union sock_arg_t *sock_args, int role);
```

Algorithm 4.5: Cleanup of socket IPC.

4.2 Write API

The write API consists of a single, versatile function: sock_writev. See algorithm 4.6 for its definition.

It requires four arguments:

- A pointer to an instance of the structure sock_writer_arg_t which will be introduced shortly.
- A pointer to an array of iovec structures. Each iovec structure defines separate memory regions of a variable size, acting as a buffer. An entire array of such structures represent a vector of memory regions[14].
- The integer invalid_count represents the number of log messages located in the iovec array. Each entry represents an attempt to establish an unwanted connection request from a malicious client.
- Finally, the maximum number of receiving sockets is given via the parameter maxNumOfSocks and is usually equal to MAX_AMOUNT_OF_SOCKETS. Setting maxNumOfSocks to a value smaller than MAX_AMOUNT_OF_SOCKETS results in that writer process only supplying a subset of sockets/readers with data.

```
int sock_writev(struct sock_writer_arg_t *sock_args,
                struct iovec *log_iovs,
                uint16_t invalid_count,
                uint16_t maxNumOfSocks);
```

Algorithm 4.6: Write API for the unix domain socket architecture

The structure sock_writer_arg_t contains all information needed by the writer process, as seen in 4.7.

```
struct sock_writer_arg_t
{
    char socketPathNames[MAX_AMOUNT_OF_SOCKETS][
       SOCKET_TEMPLATE_LENGTH];
    struct sockaddr_un socketConnections[
       SOCKET_TEMPLATE_LENGTH];
    int socketRecvs[MAX_AMOUNT_OF_SOCKETS];
    int writeSockets[MAX_AMOUNT_OF_SOCKETS];
};
```

.

Algorithm 4.7: Writer structure containing critical information being reused over several calls of sock_writev

The first parameter socketPathNames is an array containing all possible paths in which unix domain sockets could be located. Here, the necessity for defining the variables MAX_AMOUNT_OF_SOCKETS and SOCKET_TEMPLATE_LENGTH becomes evident. The entirety of the socket IPC is implemented as a static library. Consequently, arrays cannot be assigned a variable length during runtime. Another array, socketConnections, contains a collection of sockaddr_un structures. Each of these represents a single unix domain socket and their address information. The array socketRecvs stores integers displaying which sockets have already been connected to. Sockets can either be marked as not available (-1), available but not connected yet (0), or available and connected with the writer process (1). Lastly, clientSockets holds a collection of file descriptors referring to each connected socket, as created by the function socket.

When calling sock_writev, the first thing being performed is a check for newly available unix domain sockets. This can be considered analogous to checking for new readers because of the strict one-to-one mapping between sockets and readers. If new sockets were identified, a connection with that socket is established and saved for future calls of this function. Then, all sockets with an existing connection are sent the data located in the iovec structures using the blocking function write. On success, the function returns the number of sent messages via the socket IPC.

4.3 Read API

The function sock_readv is responsible for reading data out of the unix domain socket infrastructure. As seen in 4.8, the function takes two arguments.

```
int sock_readv(struct sock_reader_arg_t *sock_args,
                    struct iovec *iovecs);
```

Algorithm 4.8: Read API for the unix domain socket architecture

The parameter iovecs is a pointer to an array of iovec structures. Any log messages received via the socket IPC are stored here for the calling reader process to access later. A structure containing all relevant information regarding the specific unix domain socket associated with the reader process are stored in sock_args. This structure, sock_reader_arg_t, is defined in 4.9.

Analogous to sock_writer_arg_t, the path of the socket assigned to the reader process is passed in socketPathName. Parameter address contains the structure sockaddr_un representing that same unix domain socket. Not wanting to redetermine the static size of the address for each function call, sizeOfAddressStruct is passed along containing that exact value. Therefore, the size of address has to determined only once when initializing communication, saving computational time. The integer readSocket contains the file descriptor referring to the readers unix domain socket, as created by the function socket. Saving already established connections with writer processes for future function calls is done in the array clientSockets.

```
struct sock_reader_arg_t
{
        char socketPathName[SOCKET_TEMPLATE_LENGTH];
    struct sockaddr_un address;
        int sizeOfAddressStruct;
    int readSocket;
        int clientSockets[MAX_AMOUNT_OF_SOCKETS];
};
```

.

Algorithm 4.9: Reader structure containing critical information being reused over several calls of `sock_readv`

Calling `sock_readv` creates a list of clients which the blocking function `select` will regularly poll.

`select` waits for at least one file descriptor (analogous to: unix domain socket) to become ready for an I/O operation. A file descriptor is considered ready once a call of `read` or `write` will not block if performed.[15]

This stops the function `sock_readv` having to either be stuck in a blocking call of `read`, or return from a non-blocking call of `read` with an error code. Having a blocking call of `select` instead of `read` is desirable because it allows `sock_readv` to accept connections of new writer processes while waiting for data to arrive. Once `select` returns, `sock_readv` checks if new connections need to be accepted. If not, one of the already connected writer processes has sent data via the unix domain socket, which is ready to be read. All received data is then saved in the provided parameter `iovecs`, allowing the calling process of `sock_readv` to access it. The function `sock_readv` will then terminate and return the number of received messages.

5 Experiments

Paul Raatschen has performed a study[3] concluding that the original Fail2ban process can be improved upon. It was determined that especially with many clients, Fail2ban struggled to keep up with high incoming traffic rates. To remedy this issue, a more performant program, Simplefail2ban, was implemented and measured. An increase in performance was evident. Simplefail2ban supported two modes of Inter-process communication (IPC). The disk/file mode was akin to traditional file logging, while the shared memory approach would use a shared memory section to exchange data between processes. A direct comparison between the already outlined socket approach and previously supported IPC types necessitates the measurements in this chapter.

The following chapter details the measurements performed, outlining specifics according to Jain's "The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley"[16] chapter 2.2.

5.1 Test environment

Two machines, both identical in hardware and software, were used in these experiments. The first machine, the Device under Test (DUT), ran Simplefail2ban and a test application responsible for receiving incoming traffic and reporting clients. The second machine generated and sent traffic, consisting of both valid and invalid traffic, to the DUT using TRex.

5.2 Experimental design

In his thesis[3], Paul Raatschen showed that the shared memory mode of Simplefail2ban outperforms the traditional Fail2ban. However, it remains unclear if the implementation of this IPC type is more performant than other alternatives. Specifically, the possibility of using Unix domain sockets as a mode of inter-process communication was not explored. The following experiments enable a direct comparison between the two IPC types.
In general, the experiments consist of two participants and a one-sided data exchange. The DUT, or more specifically the application udp_server, receives a stream of both wanted and unwanted data. Identifying desired traffic is done by analyzing the message payload. This is a crude and unrealistic approach to filtering malicious communication requests. Such a simplification allows the application udp_server to quickly generate log messages. Since the goal of this study is to determine the most efficient IPC type for Simplefail2ban, it is very plausible that this abstraction does not diminish the findings of this thesis.
To compare the differing IPC types, a set of performance metrics needs to be established:

Performance metrics

- Total number of unwanted requests dropped (number of packets)
- Total number of unwanted requests dropped, relative to the total amount of unwanted requests sent (percentage)
- Number of log messages processed by Simplefail2ban, relative to the number of log messages sent by the test server (percentage)
- CPU utilization of Simplefail2ban (seconds of CPU time)

Higher is better for the first three metrics. The last metric should be minimized for the DUT so its services are continually provided to valid clients.

The fixed parameters for each of the experiments are the following:

Fixed parameters

- Hardware and Software parameters of the testbed in this table:
  - CPU: 16 cores, no hyper-threading enabled
  - Network Interface Card (NIC): Maximum transfer unit is 1500 bytes
  - TRex: One interface, 30 threads
- Number of entries in extended Berkeley Packet Filter (eBPF) maps for IPv4 & IPv6: 1M
- Number of receiving threads used by udp_server: 16
- Duration of measurement: 300 Seconds

Table 5.1: Table of Hardware and Software parameters of the testbed. The first machine serves as the DUT. The second machine generates traffic to be sent to the DUT via TRex.

Hardware	
CPU	16 × Intel(R) Xeon(R) Silver 4314 Central Processing Unit (CPU) @ 2.40GHz
NIC	Mellanox Technologies MT2892 Family [ConnectX-6 Dx]
RAM	128 GB
Software	
OS	Debian GNU's Not Unix (GNU)/Linux 11
Kernel	5.10.0-28-amd64 x86_64
NIC Driver	mlx5_core; Version 5.8-2.0.3
TRex	2.99 (Stateless)

- Amount of valid traffic sent: 50000 Packets per second (PPS)
- Number of clients sending valid traffic: 254
- **Simplefail2ban** parameters:
  - Number of hash table bins used: 6000011
  - Ban threshold for clients: 3
  - Ban time for clients: 30 seconds
  - Enabling the Regular Expression (Regex) Matching feature of Simplefail2ban (the current implementation does not ban clients correctly when disabled)
  - For **shared memory** specifically:
    - Number of banning threads used: 16
    - Line count for the shared memory buffer segments: 1M
    - Segment count for the shared memory buffer: 16
    - Overwrite feature enabled
    - Workload stealing feature disabled
  - For **sockets** specifically:
    - Number of banning threads used: 16
    - Number of sockets: Same as number of reader processes (either one, or two when utilizing a second reader)
    - Using default path to sockets created by the application: `tmp/`
    - Using default socket receive and send buffer size configured on the system: 212992 Bytes
  - For **file** specifically:
    - Number of banning threads used: 1 (file mode only supports one banning threads)
    - Buffer size for uring_getlines: 2048

The factors, or variable parameters, during these experiments were the following:

Factors and their levels

- Internet Protocol (IP) stack: IPv4, IPv6 and IPv4/IPv6 mixed
- Effects of differing amount of invalid traffic sent: 100k, 1M, 10M, 20M, 30M PPS
- Effects of differing number of clients sending invalid data: 65534 and 131068
  - Range used for 65534 clients: 10.4.0.1 to 10.4.255.254 resulting in clients stemming from 256 subnets (using offset_fixup of 5 for IPv6 in TRex script).

  - Range used for 131068 clients: 10.4.0.1 to 10.5.255.252 resulting in clients stemming from 512 subnets (using offset_fixup of 5 for IPv6 in TRex script).
  - When using the IPv4/IPv6 IP stack, the range for 65534 client is being used twice to generate both a IPv4 and IPv6 stream.

- Differing IPC type: FILE (traditional file-based logging), SHM (using shared memeory), SOCK (using unix domain sockets)
- For shared memory specifically:
  - No 2nd Reader/ Enabling 2nd Reader
- For sockets specifically:
  - No 2nd Reader/ Enabling 2nd Reader

To generate the traffic being sent to the DUT, TRex scripts are used. These scripts provide the option to modify the sent traffic according to the factors outlined above. During these measurements, adapted versions of Paul Raatschens[3] scripts were used. To measure most performance metrics, an adaptation of the xdp_ddos01_blacklist_cmdline program was used. This application originally stems from Florian Mikolajczak master's thesis[2] and routinely polls the number of dropped and passed packets from a specific eBPF map. It was modified by Paul Raatschen to output values as a comma-separated values (csv) file. The polled eBPF map is ultimately used by Simplefail2ban to ban clients. CPU time was measured via the command top.

5.3 Established Simplefail2ban IPC types

Software version changes warrant remeasurement of the shared memory and file IPC mode for Simplefail2ban. These will also be used to evaluate the newly implemented socket mode.

5.3.1 Experiment 1a: Simplefail2ban Logfile

It has already been shown that the file IPC type of Simplefail2ban is outperformed by the shared memory mode. Pure IPv4, IPv6 and a mixed IPv4/IPv6 IP stack will be used. File logging is expected to perform worse than the other IPC types discussed in this thesis, and acts as a baseline for the other IPC types. In total, 25 unique measurements were conducted for this experiment.

5.3.2 Experiment 1b: Simplefail2ban Shared Memory

The newly implemented socket approach is intended to be a valid alternative to the shared memory mode of Simplefail2ban. To enable a direct comparison, measurements for the shared memory mode need to be done under high enough loads, since with lower loads both the socket and shared memory mode are suspected to be performant enough. All levels of invalid traffic rates are measured individually. Again, either a pure IPv4, IPv6 or mixed IPv4/IPv6 IP stack is utilized. The most performant features will be used, meaning overwrite is enabled and workload stealing is disabled. No second reader process is being employed here. In total, 25 unique measurements were conducted for this experiment

5.4 Measuring the socket Application Programming Interface (API)

In the following section, thorough variations of factors and their levels are used to evaluate the performance of the socket mode. Also, heavy workloads are employed to determine how the socket mode performs in worst case scenarios. This will allow for a direct comparison between socket and shared memory mode. The data flow in the DUT can be seen in 5.1.

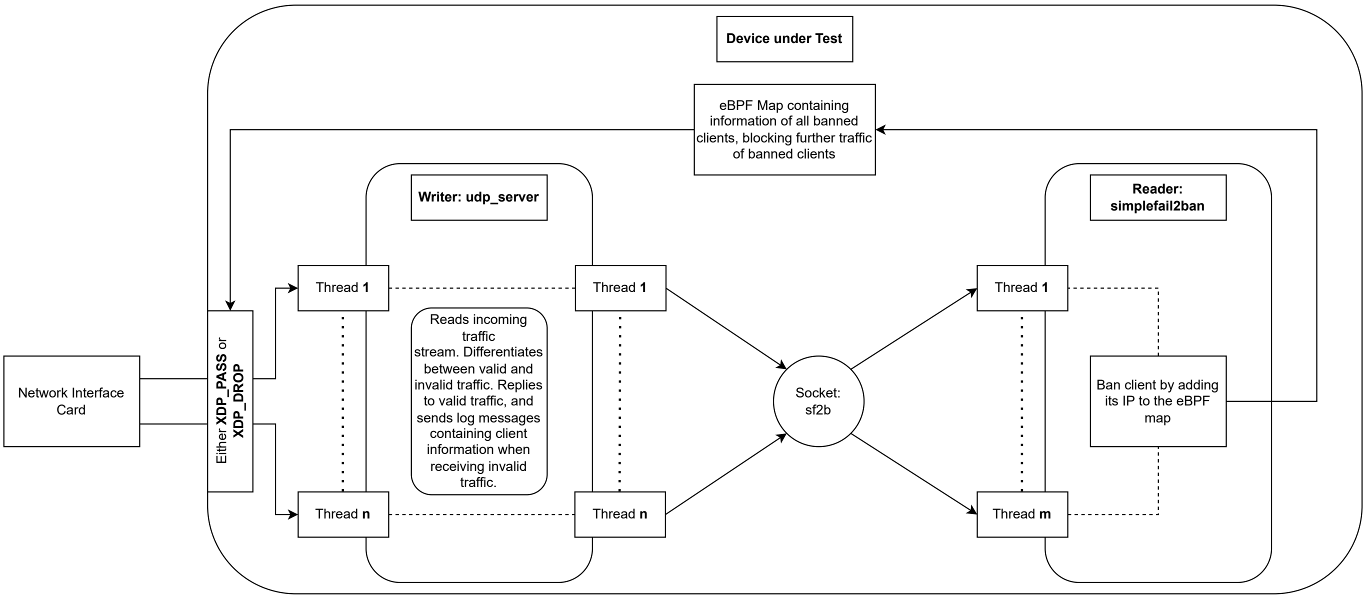

Device under Test
eBPF Map containing information of all banned clients, blocking further traffic of banned clients
Writer: udp_server
Reader: simplefail2ban
Network Interface Card
Either XDP_PASS or XDP_DROP
Thread 1
Thread n
Thread 1
Thread n
Reads incoming traffic stream. Differentiates between valid and invalid traffic. Replies to valid traffic, and sends log messages containing client information when receiving invalid traffic.
Socket: sf2b
Thread 1
Thread m
Ban client by adding its IP to the eBPF map

Figure 5.1: displays the data flow (left to right) on the DUT when enabling sockets as the IPC type. A packet can either be passed (XDP_PASS) to the kernel or dropped (XDP_DROP) before ever reaching the kernel.

5.4.1 Experiment 2: Simplefail2ban Sockets

To establish a baseline for the performance of the socket mode, all factors are set to all possible levels in every combination. The only exception being the possibility of using a second reader process, which will have its own section later. In total, 25 unique measurements were conducted for this experiment.

5.4.2 Experiment 3a: Replication of Simplefail2ban Shared Memory with 2nd Reader

In order to later compare the socket mode and its option to have a second reader, a baseline measurement needs to be established. This experiment will be performed with 131068 clients sending invalid data only and no pure IPv4 or IPv6 IP stack. Again, the overwrite feature is enabled while workload stealing is disabled. The second reader process logs all messages it receives in a log file. In total, 5 unique measurements were conducted for this experiment.

5.4.3 Experiment 3b: Simplefail2ban Sockets with 2nd Reader

This experiment closely mirrors the experiment 3a. A total of 131068 clients will send invalid data to the DUT with no pure IPv4 or IPv6 IP stack. The shared memory mode inherently supports the possibility of adding a second reader to the shared memory section to read log messages. There is no such inherent support in the socket mode. Instead in its current implementation, another read process can be started which will then be assigned its own socket. This socket will then also receive all log messages. Consequently, the shared memory mode will likely see a smaller performance loss, since no additional effort is required to send messages to the second reader. The second reader process logs all messages it receives in a log file. In total, 5 unique measurements were conducted for this experiment.

5.5 Evaluation of Experiments

The aforementioned experiments can be logically grouped in two categories: Baseline measurements and utilization of a second reader. Baseline measurements are conducted in experiments 1a, 1b and 2, second reader experiments in 3a and 3b. With 85 performed experiments, a thorough yet not unreasonably long evaluation of each measurement is impossible. Instead, only especially expressive data will be covered in this section, with any notable or diverging observations being explicitly mentioned. For full measurement data on all experiments, refer to the repository provided in the sources[17].

Meaning of data variables

In the following section, each graph will be accompanied with an additional table. This table contains data that is not explicitly expressed otherwise. A total of six lines are plotted, with two of them belonging to each IPC type. For each IPC type (File, Shm, Sock), the total number of packets dropped by the eBPF program is denominated via `XDP_DROP`. Similarly, the number of packets passed to the kernel is displayed via `XDP_PASS`. The `relative drop` represents the percentage of packets dropped relative to the theoretical maximal of dropped packets (represented by `Best-case drop rate`). Calculating the relative drop is done with the following formula: total dropped packets /(experiment duration $*$ invalid traffic rate $-$ number of ban cycles $*$ ban limit $*$ number of malicious clients). Each ban cycle lasts for 30 seconds. `Packets received by udp_server` lists the number of packets reaching the application udp_server, while `Log messages` represents the number of messages sent via the chosen IPC type.

5.5.1 Baseline measurements

For this section, data of 75 experiments have been analyzed.

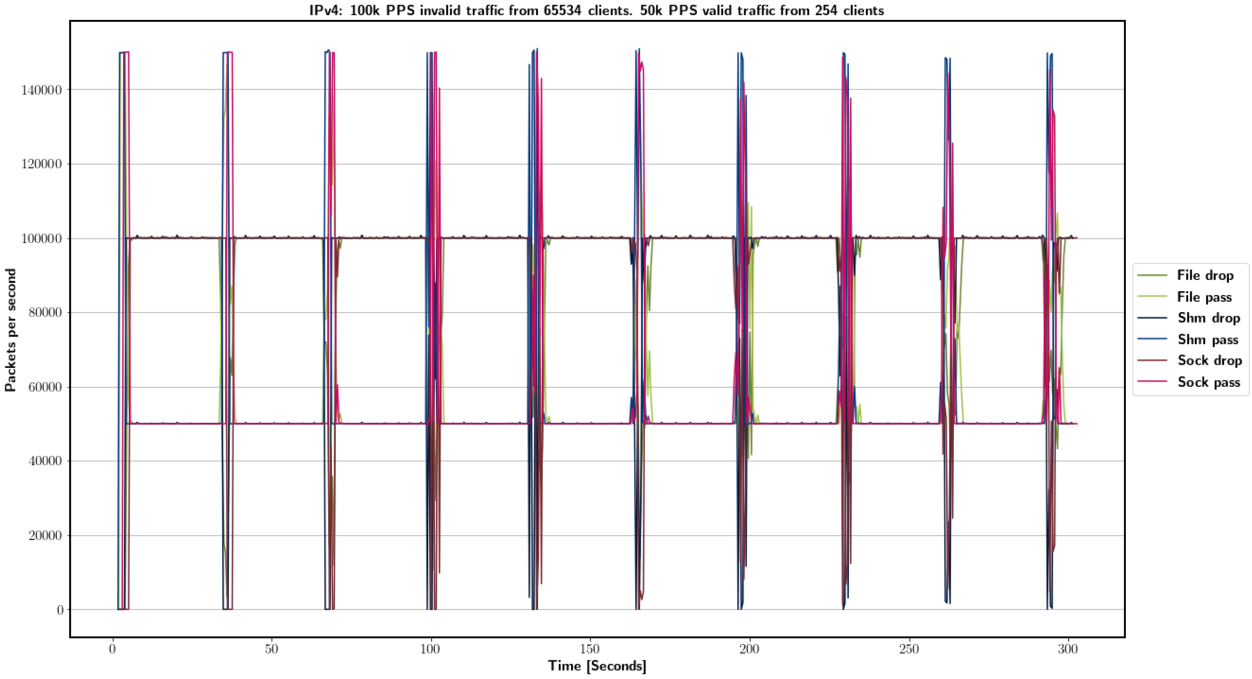
Using 65534 clients to send invalid data

A trend displayed in 5.2 remains prevalent with lower rates of invalid traffic with 65534 clients sending invalid data: Differences in performance are difficult to spot when graphed. The table in 5.2 does provide more information. All IPC types perform similarly well in defending the Denial-of-service (DoS) attack.

However, one anomaly stands out. The `relative drop` of the shared memory IPC type is over 100%. This happened a total of 21 times over separate measurements, but only when measuring traffic rates of 100k PPS. It is suspected that TRex is unable to provide a reliable stream of packets, but only when starting each measurement call. Once a steady stream of packets at desired rates has been established, it does remain stable. Therefore, the number of packets that should be received are higher than the actual incoming traffic rate, resulting in slight inaccuracies when calculating the `relative drop`. While not directly apparent at 100k PPS due to slight rounding, this discrepancy can be detected by totaling `XDP_DROP` and `XDP_PASS` up and comparing it to the number of packets that should have theoretically been received. Unfortunately, this issue is present in all data presented in this thesis and depicts a likely flaw in TRex - not the measurements.

Of note is a stark difference in CPU time, with a clearly discernible spike when using the socket IPC type, likely cause by extensive context-switches between kernel- and userspace.

All IPC types have sent the same number of log messages indicating that, at lower traffic rates, all clients can be banned with the first message they send surpassing the ban limit, regardless of IPC type. This claim is supported by the fact that the log messages coincide with the formula: number of ban cycles ∗ ban limit ∗ number of malicious clients; which is the theoretical minimum number of messages needed to successfully ban all clients throughout the duration of the measurement. Otherwise, the results are as expected with the file IPC type performing worst due to increased latency.



IPC type	XDP_DROP [10^6]	XDP_PASS [10^6]	Relative drop [%]
File	27,84	17,16	99,29900071
Shm	28,03	16,97	100,0000036
Sock	28,02	16,98	99,93706566

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	16,81	19,66	04.95
Shm	16,97	19,66	05.94
Sock	16,96	19,66	57.90

Figure 5.2: Total packets sent: 45m. Best-case drop rate: 93,4466%

In 5.3, the trend of shared memory clearly outperforming other IPC types is evident. Again, differences in plotting the number of dropped and passed packets are to small to be visually decipherable and will be omitted until this changes. Instead, a clear difference in performance is once again mainly visible in CPU time. The `relative drop` also reveals that the shared memory IPC is performing best, with a lower number of packets being passed to the kernel. An overall drop in performance measured via `relative drop` along all IPC types was expected. A generous best-case scenario consists of assuming that all 65534 clients are banned in the same timespan at all incoming traffic rates. But even then, the inherit latency (even if miniscule) of the Intrusion Prevention System (IPS) means that an increase in invalid traffic flow directly correlates with more packets reaching the kernel during each ban cycle. Therefore, the `relative drop` rate must inversely correlate with the invalid traffic rate.

IPC type	**XDP_DROP [10^8]**	**XDP_PASS [10^6]**	**Relative drop [%]**
File	29,40	75,07	98,05973977
Shm	29,75	37,49	99,21848407
Sock	29,50	61,81	98,39458207

IPC type	**Packets received by udp_server [10^6]**	**Log messages [10^5]**	**CPU [seconds]**
File	17,51	35,10	09.69
Shm	20,02	51,93	16.86
Sock	17,13	25,52	76.00

Figure 5.3: Total packets sent: 3015m. Best-case drop rate: 99,934466%

Even with 30m invalid PPS, as seen in 5.4, all IPC types successfully defend against the DoS attack. The fact that the file IPC type outperforms the socket IPC type in terms of `relative drop` rate is especially noteworthy. File mode also outperforms shared memory and socket mode on CPU usage, which was not expected due to the traditionally high latency of file-based logging approaches. With the high rate of incoming traffic during a ban cycle, the application udp_server has to wait on the IPC to be able to submit more log messages to Simplefail2ban. This results in the system being unable to submit a substantial number of packets to udp_server. While not explicitly documented in this thesis, these numbers are available in the repository[17]. The difference in packets received by udp_server and number of passed packets correlates exactly with this observation. Measuring the number of packets unable to be submitted to udp_server was done by checking the file `/proc/net/udp6` on the DUT.

A mutual feature of these three measurements is the direct correlation between `relative drop` rate and number of `packets received by udp_server`. Better performing IPC types generally receive more packets despite the fact that they block invalid traffic at an increased rate. This is explainable through the inability of the system to supply new packets to udp_server while it is still waiting on the IPC architecture to deliver data to the IPS. IPC types with lower latency and higher bandwidth can log more messages during the short influx of packets during each ban cycle. Hence, udp_server can receive more packets from the system. These attributes also result in higher `relative drop` rates, thus causing this observation.

Figure 5.5 displays IPC types under 30m invalid PPS from 65534 different clients using IPv6 addresses in contrast to the previously used IPv4 addresses. Usage of different IP stacks had little impact on the overall results of this thesis, which is why lower traffic rates for IPv6 are omitted.

As expected, the file IPC type performed worst, with shared memory performing best.

IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
File	87,75	159,82	97,52375345
Shm	88,30	87,23	98,13105047
Sock	87,45	139,42	97,18179422

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	17,48	40,72	16.55
Shm	21,39	69,92	39.08
Sock	16,92	31,62	138.85

Figure 5.4: Total packets sent: 9015m. Best-case drop rate: 99,97815533%

However, contradicting expectations, the socket and shared memory IPC types performed better when dealing with IPv6 instead of IPv4 addresses. This is surprising since no changes in general behavior are expected in either udp_server or Simplefail2ban with one exception: The Regex matching feature. In theory, IPv6 banning should be less performant than using IPv4, because IPv6 addresses are longer than IPv4 addresses. Reading these addresses out of the log messages received by Simplefail2ban using the Regex matching feature should require more time. This idea is supported by the increase in CPU time across all IPC types compared to figure 5.4 using IPv4. However, an increase in performance can be measured in almost all experiments using IPv6 instead of IPv4. What exactly causes this abnormality is unclear.

IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
File	87,41	211,05	97,14091697
Shm	88,63	85,55	98,50239609
Sock	87,77	170,03	97,54838057

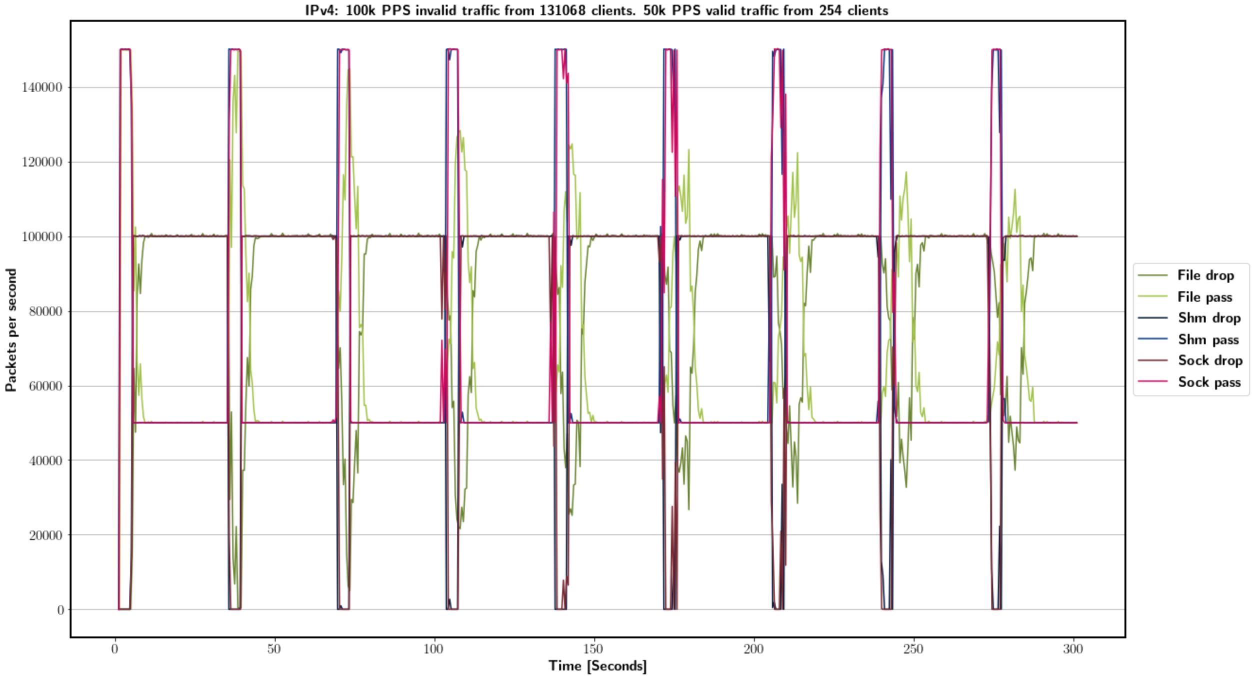
IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	17,20	38,73	22.51
Shm	21,79	72,38	46.03
Sock	16,92	30,04	149.69

Figure 5.5: Total packets sent: 9015m. Best-case drop rate: 99,97815533%

Using 131068 clients to send invalid data

When using 131068 clients to send invalid data, changes in performance become visible when plotted. Figure 5.6 displays only 100k invalid PPS, yet the file IPC type struggles to quickly ban all clients. In contrast, both socket and shared memory seem to perform quite similarly across all statistics except for CPU time. Here, the socket IPC type occupies the CPU almost ten times longer than shared memory, again likely due to extensive context-switches between kernel- and userspace.

At invalid traffic rates of 10m PPS, displayed in figure 5.7, drastic changes are noticeable. Firstly, shared memory does outperform both the socket and file IPC type in `relative drop` rate. Also, the latency of each IPC type is clearly visible in the provided graph via their drasticly shifted spike of dropped packets each ban cycle. The shared memory IPC type both starts and ends its ban cycles before the socket and file IPC type. Unexpectedly, the socket IPC type still outperforms the file IPC type in `relative drop` rate. A general



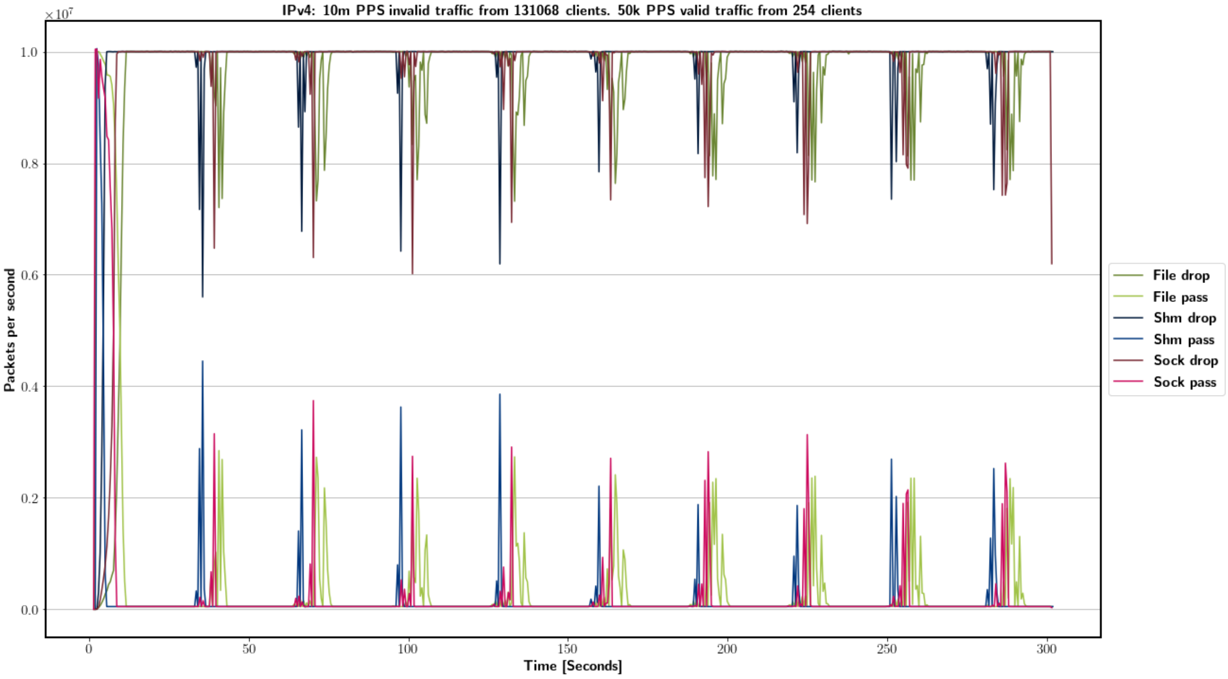
IPC type	XDP_DROP [10^6]	XDP_PASS [10^6]	Relative drop [%]
File	25,99	19,01	99,69409958
Shm	26,46	18,54	101,5083842
Sock	26,44	18,56	101,4395334

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	18,16	35,39	08.34
Shm	18,54	35,39	10.14
Sock	18,53	35,39	100.40

Figure 5.6: Total packets sent: 45m. Best-case drop rate: 86,8932%

delay of each ban cycle is visible too. The second to last ban cycle, which should start at 240 seconds, starts late at about 250 seconds. This delay is present in all IPC types. It is likely caused by the unbanning thread of Simplefail2ban having to handle 131068 clients, resulting in them being banned, on average, slightly longer than the intended 30 seconds.

At 30m invalid PPS, seen in figure 5.8, the file IPC type starts and finishes its ban cycles later than other IPC types. While the shared memory and socket IPC type start almost simultaneously, sockets require longer to finish a full ban cycle. Differences in CPU time are prominent, with the socket IPC performing worst and file IPC performing best according to the `relative drop` rate. A general trend found in all measurements presented is the low throughput of the socket IPC type. Here, it is especially pronounced with the shared memory IPC type managing to transmit almost double the amount of `log messages`.



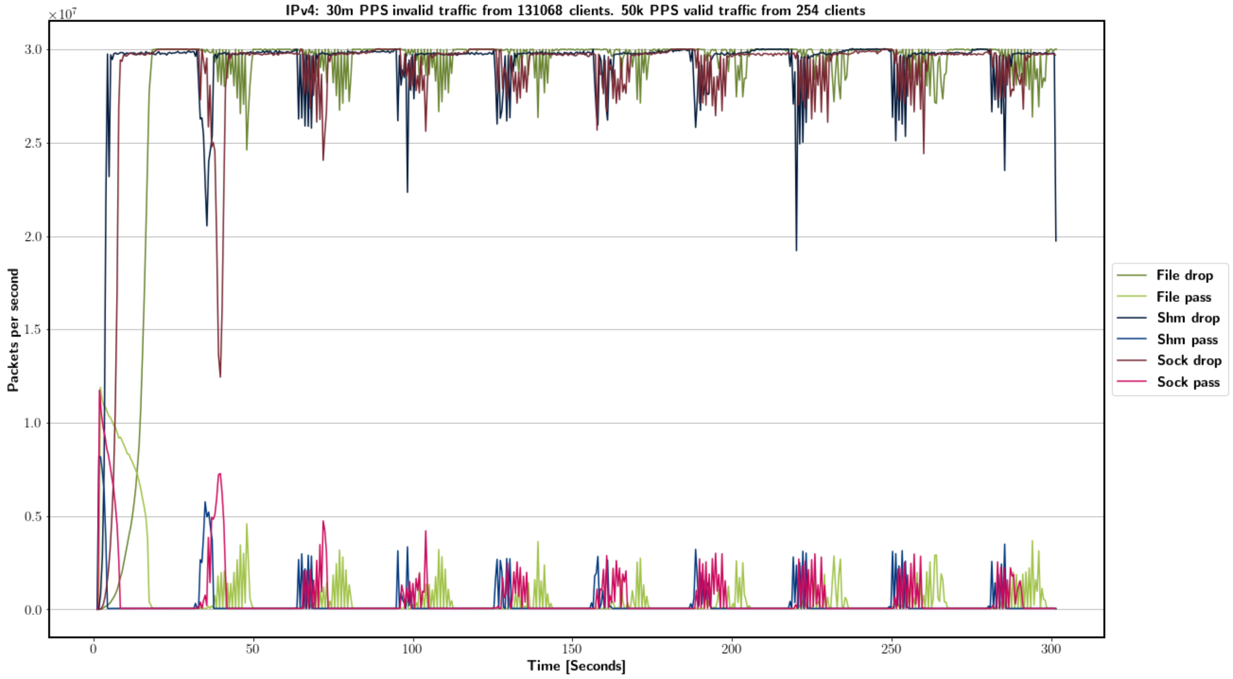
IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
File	28,77	136,15	96,03486581
Shm	29,54	58,36	98,60515631
Sock	29,16	95,52	97,33669129

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	19,31	63,90	19.39
Shm	25,40	107,54	29.47
Sock	19,02	47,67	133.54

Figure 5.7: Total packets sent: 3015m. Best-case drop rate: 99,868932%

Switching to IPv6 at a rate of 30m invalid PPS does not yield any fundamentally different results: 5.9. Again, performance of all IPC types improves slightly, still with no known cause. Shared memory performs best, with the file IPC type performing worst. The direct correlation between `relative drop` rate and `packets received by udp_server` is also still present.

When using a mixed IP stack, two streams of invalid data were configured in TRex: One being IPv4, the other being IPv6. Both streams were producing exactly 50% of the invalid traffic and consisted out of 65534 clients each. Naturally, no single client sent both IPv4 and IPv6 packets. The most expressive measurement is displayed in 5.10 with 30m invalid PPS. Expectations were that the `relative drop` rate for all IPC types would land firmly between the equivalent measurements using IPv4 and IPv6. However, this is not



IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
File	85,02	238,30	94,51036756
Shm	87,57	104,14	97,33826458
Sock	86,12	180,89	95,73084169

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	18,04	74,39	38.99
Shm	25,32	115,04	71.92
Sock	18,33	59,26	323.02

Figure 5.8: Total packets sent: 9015m. Best-case drop rate: 99,95631067%

IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
File	85,73	228,07	95,29278185
Shm	87,60	109,08	97,37706621
Sock	86,21	177,33	95,82614459

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	17,90	69,14	38.41
Shm	25,08	111,45	74.71
Sock	18,67	61,84	317.37

Figure 5.9: Total packets sent: 9015m. Best-case drop rate: 99,95631067%

the case. The figure 5.10 does not display this property. In fact, no other measurements using a mixed IP stack definitively exhibit this property. Here, in figure 5.10, the shared memory and socket IPC types outperform the measurement utilizing pure IPv6 displayed in 5.9. Though, most measurements still outperform their pure IPv4 counterpart at a minimum. But performance regarding their pure IPv6 counterpart is not definitive, the variance between measurements is too severe.

IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
File	85,12	286,15	94,61335186
Shm	88,02	105,83	97,84149307
Sock	86,30	212,81	95,93428297

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
File	17,69	70,65	47.15
Shm	25,13	111,62	94.64
Sock	18,00	59,85	353.34

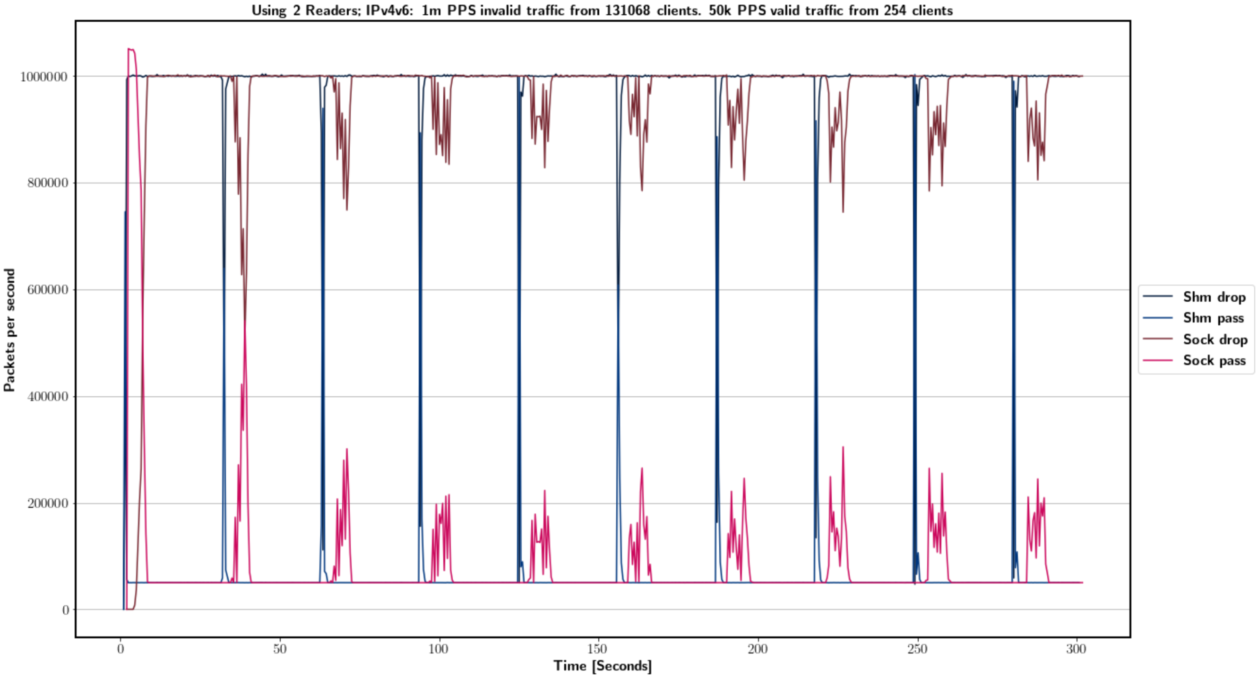
Figure 5.10: Total packets sent: 9015m. Best-case drop rate: 99,95631067%

5.5.2 2nd reader measurements

Initially, data of 10 experiments have been analyzed for this section. At 100k invalid PPS, performance of both the shared memory and socket IPC types was almost identical. The only exception is the CPU time in which the socket IPC performed up to six times worse than shared memory. The results become interesting when increasing invalid traffic rate to 1m PPS, a seen in figure 5.11. The graph displays a clear delay between the ban cycles of the shared memory and socket IPC types. Still, both modes are able to supply both the IPS Simplefail2ban and a slower reader process with data while defending against the DoS attack. Again, CPU time of the socket IPC type is significantly higher than its shared memory counterpart. The `relative drop` rate is also worse by about 2 percent (equal to 6,3M more packets reaching the kernel over the duration of the measurement), with fewer messages logged.

Figure 5.12 shows data measured with 20m invalid PPS. The socket IPC type struggles to supply both the IPS and the second reader with `log messages`, having logged less data than the shared memory IPC type. The graph also displays this inability to keep up with incoming traffic during the first two ban cycles. While shared memory was able to fully ban all malicious client in just one 30 second ban cycle, the socket mode is not. Given enough time, the socket IPC type is able to recover and successfully defend against the DoS attack.

That changes in figure 5.13. Now, the socket IPC type is unable to defend against the DoS attack, and the system is overwhelmed - failing to drop all invalid traffic. The `relative drop` falls to approximately 54 percent, the number of messages passed to the kernel rise significantly and the application udp_server is not able to handle the influx of incoming data. Fewer log messages are sent to all readers and the CPU time falls, likely due to the system not having any resources left for user space applications. Meanwhile, the shared memory IPC type performs just as well as in figure 5.10, when no second reader



IPC type	XDP_DROP [10^7]	XDP_PASS [10^6]	Relative drop [%]
Shm	29,53	19,75	99,72283593
Sock	28,91	25,94	97,6334018

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
Shm	19,48	44,91	17.76
Sock	18,29	41,47	80.82

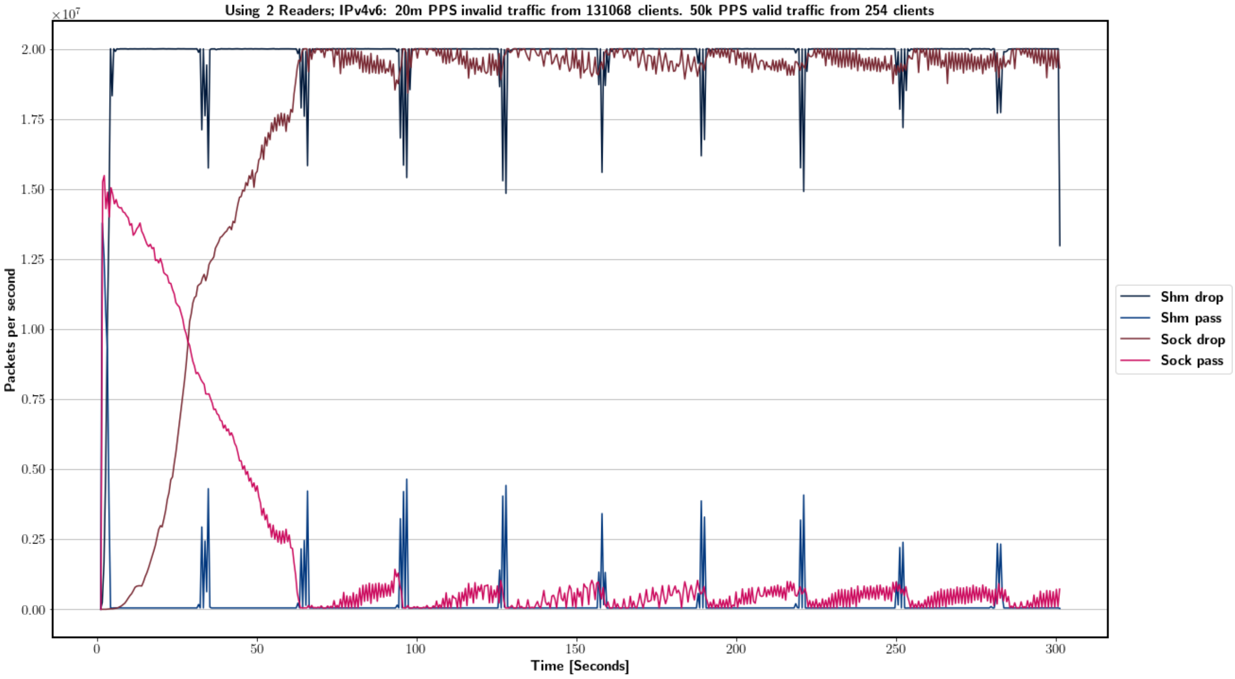
Figure 5.11: Total packets sent: 315m. Best-case drop rate: 98,68932%

was attached to the IPC architecture.

A difference of this scale was not initially expected. However, increased performance of the shared memory IPC type was attributed to the overwrite feature. Enabling this feature meant that slower reader processes were ignored if they slowed down any writers. The socket IPC type cannot abandon slow readers, it has to wait for each reader to receive all data, unlike the shared memory IPC.

Another experiment was conducted to confirm the link between enabling the overwrite feature and the increased performance of the shared memory IPC type: The shared memory IPC type is used without having the overwrite feature enabled. Expectations were, that having to wait for all reader processes to receive data would result in a performance decrease. This measurement is displayed in figure 5.14.

No such decrease in performance was measured whatsoever. The only logical conclusion

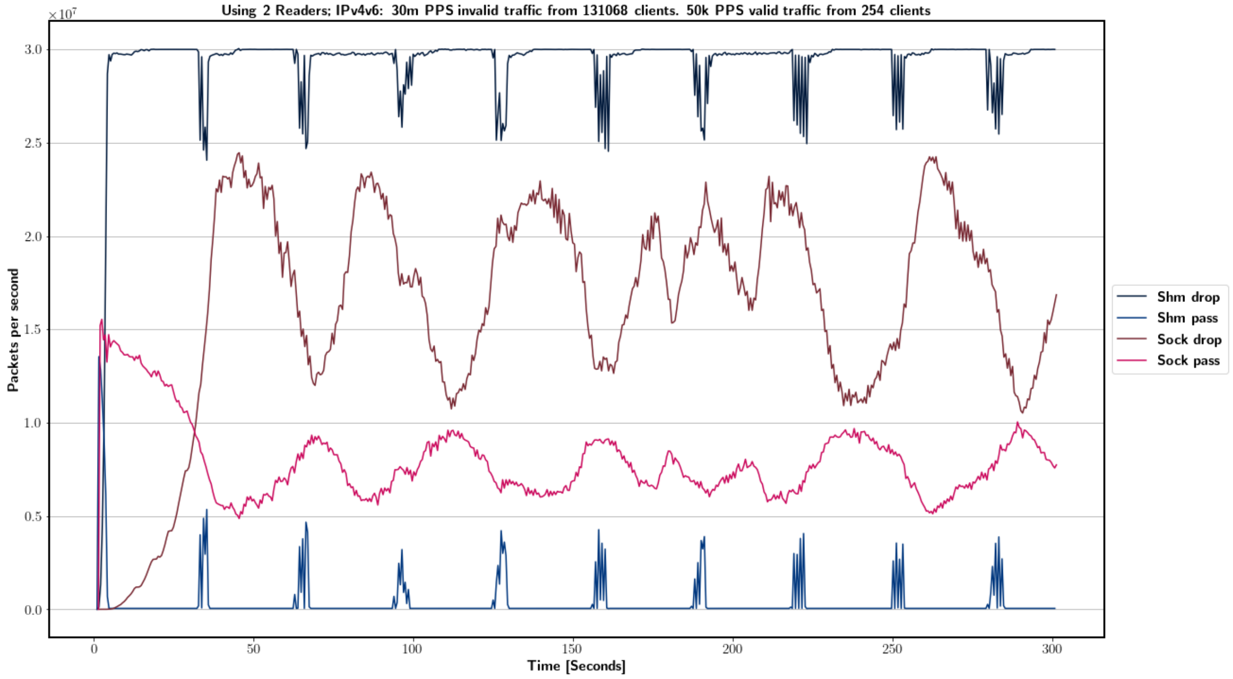


IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
Shm	59,15	79,92	98,64873119
Sock	52,45	624,81	87,47139407

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
Shm	24,61	101,26	49.90
Sock	11,32	65,01	251.49

Figure 5.12: Total packets sent: 6015m. Best-case drop rate: 99,934466%

is, that the shared memory IPC type manages to supply multiple readers with faster data transfer than is required for even 30m incoming invalid PPS. The overwrite feature was not needed and had no real impact on performance. Increased latency when using the socket IPC type likely culminated to such a degree that supplying multiple readers with data was not feasible.



IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
Shm	87,83	118,74	97,63360872
Sock	48,80	2385,53	54,22323337

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
Shm	24,70	108,17	94.60
Sock	3,10	30,04	31.91

Figure 5.13: Total packets sent: 9015m. Best-case drop rate: 99,95631067%

IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]
Shm	87,99	125,56	97,81321365
Sock	48,80	2385,53	54,22323337

IPC type	Packets received by udp_server [10^6]	Log messages [10^5]	CPU [seconds]
Shm	24,95	108,43	99.47
Sock	3,10	30,04	31.91

Figure 5.14: Total packets sent: 9015m. Best-case drop rate: 99,95631067%

6 Conclusion & Outlook

In this thesis, the previously developed light-weight Intrusion Prevention System (IPS) Simplefail2ban and its selection of Inter-process communication (IPC) was expanded to include unix domain sockets. With it being the first kernel-based IPC available, thorough measurements were conducted to evaluate its performance to the already implemented shared memory and file-base IPC modes. Expectations were that unix domain sockets would not outperform shared memory, because of a constant need for context-switches between kernel- and userspace. This initial hypothesis turned out to be true. The socket IPC type was beat in all analyzed metrics (number of unwanted requests dropped, number of log messages sent and Central Processing Unit (CPU) time) by the shared memory mode of Simplefail2ban. However, performance of the unix domain socket mode was still well up to the task. It remained competitive in experiments utilizing only one process to receive data, performing only around 2 percent worse than its shared memory counterpart. During less intense traffic flow, this small disadvantage in performance shrunk even further. Data indicates that latency in the socket IPC type was at least on par with the shared memory mode. Rather, a lack of bandwidth and increased drain on system resources are the main causes identified for the observed decrease in performance. Conversely, unix domain socket did consistently outperform the file-based IPC, albeit by a small fraction: regularly being less than one percentage point in relative drop rate. Overall, the socket IPC type was always able to block over 95,5 percent (and 98,5 percent on average) of all incoming traffic, therefore defending against Denial-of-service (DoS) attacks successfully.

While never being an explicitly desired feature, the socket IPC type does provide the option to attach, up to a pre-defined maximum, and detach a varying number of both writer and reader processes during runtime. In contrast, the shared memory IPC only provides the option to attach multiple reader processes. This allows for, in theory, flexible reusing of the socket IPC architecture for other applications. Regrettably however, usage of unix domain sockets in scenarios with multiple reader processes is not recommended due to a lack of bandwidth, resulting in the socket IPC performing significantly worse than the shared memory IPC. At a rate of 20m invalid Packets per second (PPS), defending against a DoS attack in a single ban cycle was unfeasible when employing unix domain sockets. Yet, after multiple ban cycles, Simplefail2ban was able to recover and repel the incoming DoS attack. With 30m invalid PPS, a recovery became impossible. The shared memory IPC type was able to keep its performance up even when supplying a second reader process, and disabling the overwrite feature.

Potential improvements of the socket IPC should focus on increasing the bandwidth. This makes it possible to react more efficiently and effectively in the event of a sudden influx of messages, primarily occurring at the beginning of a ban cycle or DoS attack.

Overall, this thesis proved that the kernel-based unix domain sockets remain a viable option as IPC coming reasonably close to, yet unable to surpass the higher performance of the shared memory IPC type. Continued development could focus on the possibility of scaling the socket IPC beyond the local system by employing internet sockets. Providing an improved high-level Application Programming Interface (API) for easier integrability with established real-world applications, such as syslog or journald, is also worthwhile investigating.

List of Figures

List of Tables

List of Algorithms

A Abbreviations

IDSs	Intrusion Detection Systems
HIPS	Host-based Intrusion Prevention System
HIDS	Host-based Intrusion Detection System
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
Regex	Regular Expression
TCP	Transmission Control Protocol
IP	Internet Protocol
eBPF	extended Berkeley Packet Filter
XDP	eXpress Data Path
IPC	Inter-process communication
UDP	User Datagram Protocol
DPDK	Data Plane Development Kit
CPU	Central Processing Unit
PMDs	Poll Mode Drivers
ACK	Acknowledgment
TCP-SYN	Synchronize/Acknowledge
DoS	Denial-of-service
API	Application Programming Interface
ID	Identification
DUT	Device under Test
GNU	GNU's Not Unix
NIC	Network Interface Card
PPS	Packets per second
csv	comma-separated values

B Source Files

The source files and the corresponding repository can be accessed by contacting the second supervisor: Max Schrötter.

Bibliography

[1] *Fail2ban GitHub*. https://github.com/fail2ban/fail2ban/wiki/How-fail2ban-works. 2017 (cit. on pp. 1, 4).

[2] Florian Mikolajczak. “Implementation and Evaluation of an Intrusion Prevention System Leveraging eBPF on the Basis of Fail2Ban”. MA thesis. University of Potsdam, 2022 (cit. on pp. 1, 4, 5, 20).

[3] Paul Raatschen. *Design and Implementation of a new Inter-Process Communication Architecture for Log-based HIDS for 100 GbE Environments*. Bachelorthesis. 2023 (cit. on pp. 1, 5, 6, 8–10, 17, 20).

[4] James P. Anderson. “Computer Security Threat Monitoring and Surveillance”. In: *James P. Anderson Company* (1980) (cit. on p. 3).

[5] Dorothy E. Denning. “An Intrusion-Detection Model”. In: *IEEE Transactions on Software Engineering* SE-13.2 (1987) (cit. on p. 3).

[6] Letou Kopelo, Devi Dhruwajita, and Y. Jayanta Singh. “Host-based Intrusion Detection and Prevention System (HIDPS)”. In: *International Journal of Computer Applications* 69.(0975 - 8887) (2013) (cit. on p. 4).

[7] Linux man-pages project. *Socket*. https://man7.org/linux/man-pages/man2/socket.2.html. Last visited on 09-07-2024. 2024 (cit. on p. 7).

[8] Brian "Beej Jorgensen" Hall. *Beej’s Guide to Network Programming: Using Internet Sockets*. 2023 (cit. on p. 7).

[9] Linux man-pages project. *Unix Domain Socket*. https://man7.org/linux/man-pages/man7/unix.7.html. Last visited on 09-07-2024. 2024 (cit. on pp. 7, 10).

[10] Cisco Systems. *Official TRex Website*. https://trex-tgn.cisco.com/. Last visited on 09-07-2024 (cit. on p. 8).

[11] The Linux Foundation. *Official DPDK Website*. https://www.dpdk.org/. Last visited on 09-07-2024 (cit. on p. 8).

[12] *Official Scapy Website*. https://scapy.net/. Last visited on 16-07-2024 (cit. on p. 8).

[13] IEEE Computer Society and The Open Group. *IEEE P1003.1™, Draft 3*. Tech. rep. 2007 (cit. on p. 10).

[14] Linux man-pages project. *Iovec*. https://man7.org/linux/man-pages/man3/iovec.3type.html. Last visited on 11-07-2024. 2024 (cit. on p. 14).

[15] Linux man-pages project. *Select*. https://man7.org/linux/man-pages/man2/select.2.html. Last visited on 12-07-2024. 2024 (cit. on p. 16).

[16] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley*. John Wiley & Sons, Inc., 1991 (cit. on p. 17).

[17] Daniel Aeneas von Rauchhaupt. *Thesis Git Repository*. `https://gitup.uni-potsdam.de/fips/fips_sock`. Repository containing all source code and measurement data. 2024 (cit. on pp. 23, 25).